Quality Assurance and Testing

VB CHANNELS EVENTS NEWSLETTERS Search

DEV

Microsoft announces Battle Royale Mode for Visual Studio 2019

EMIL PROTALINSKI @EPRO JUNE 6, 2018 10:58 AM

Visual Studio

Microsoft today announced Visual Studio 2019, the next version of its IDE with integrated Battle Royale mode. Release timing will be shared "in the coming months," with the company simply promising "to deliver Visual Studio 2019 quickly and iteratively." The news comes days after Microsoft's acquisition of GitHub.

VB Recommendations

Ctrl-labs' armband lets you control computer cursors with your mind

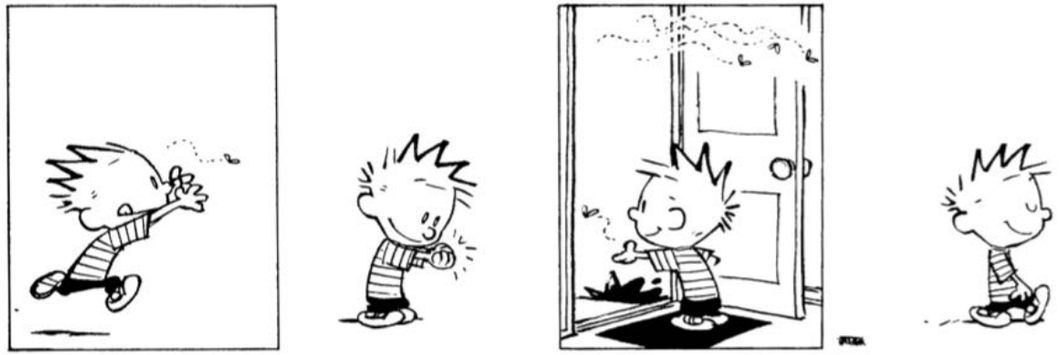
What Alienware has learned from 10 years of esports

One-Slide Summary

- **Quality Assurance** maintains desired product properties through process choices.
- **Testing** involves running the program and inspecting its results or behavior. It is the dominant approach to software quality assurance. We use:
  - regression testing to make sure new things don't break old,
  - unit testing to test individual pieces, and
  - integration testing to test everything end-to-end
- **Mocking** uses simple replacement functionality to test difficult, expensive or unavailable modules or features.

Story So Far

- We want to deliver **high-quality** software at a **low cost**. We can be more **efficient** if we **plan** and use a software development **process**.
- Planning requirements information: we **measure** the world to *combat uncertainty* and *mitigate risk*.
- But how do we measure, assess or assure software quality?



Quality Motivation

- External (Customer-Facing) Quality
  - Programs should “do the right thing”
    - So that customers buy them!
- Internal (Developer-Facing) Quality
  - Programs should be readable, maintainable, etc.

Brenan Keller
@brenankeller

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

1:21 PM · 30 Nov 18

Internal-Facing Quality

- If the dominant activity of software engineering is maintenance …
  - Then internal quality is mostly maintainability!
- How do we ensure maintainability?
  - Human code review
  - Static analysis tools and linters
  - Using programming idioms and design patterns
  - Following local coding standards
- More on this in future lectures!

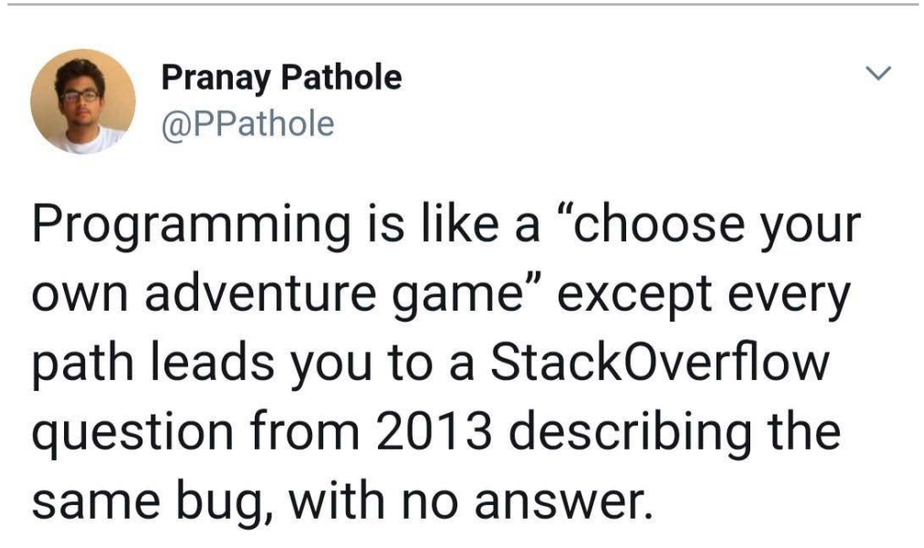
https://programming-idioms.org/

External-Facing Quality

- What does "*Do The Right Thing*" Mean?
- Behave according to a specification
  - Foreshadowing: What is a good specification?
- Don't do bad things
  - Security issues, crashing, etc.
  - Some failure is inevitable. How to handle it?
- Robustness against maintenance mistakes
  - Do "fixed" bugs sneak back into code?

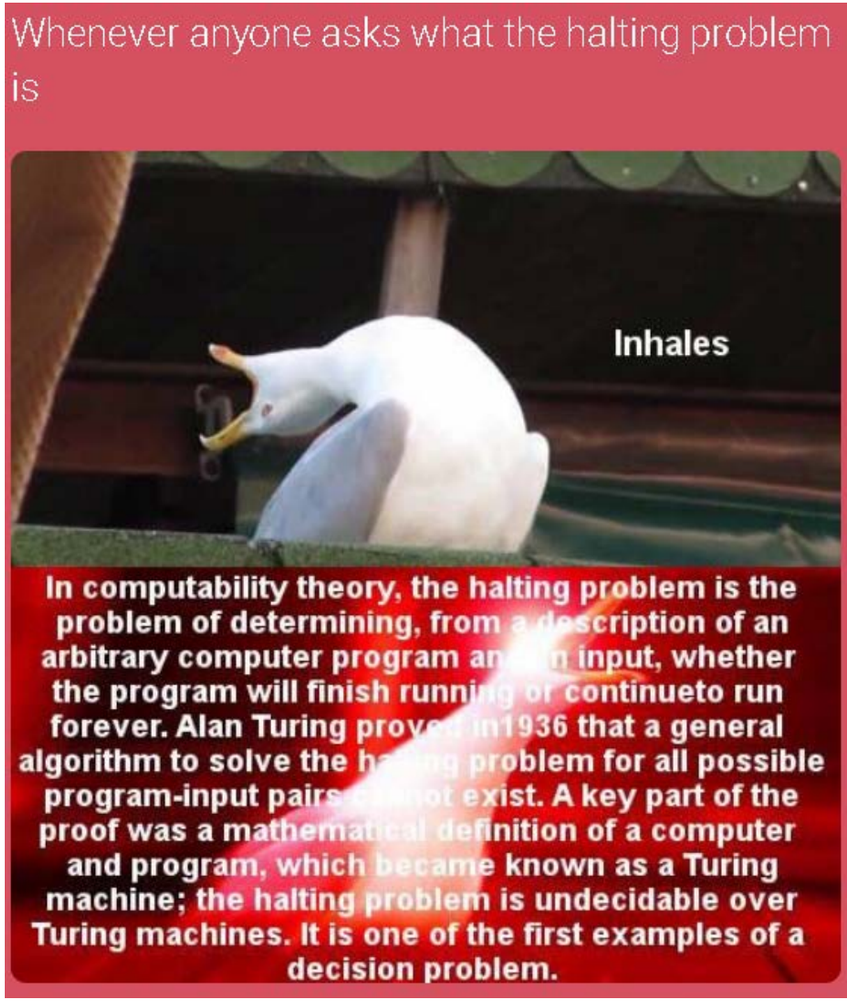
Doing The Right Thing

- Why don't we just write a new program X to tell us if our software Y is correct?



Doing The Right Thing

- Why don't we just write a new program X to tell us if our software Y is correct?
- The Halting Problem prevents X from giving the right answer every time
  - ~~X always gives a wrong answer~~
  - X cannot always give a right answer
- We can still approximate!
  - Type systems, linters, static analyzers, etc.



Practical Solution: Testing

Testing

- "**Software testing** is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test."
- A typical test involves input data and a comparison of the output. (More next lecture!)
- Note: unless your input domain is finite, testing does *not* prove the absence of all bugs.
- Testing gives you confidence that your implementation adheres to your specification.

Testing in Vandy CS Courses

- CS 1100/1101: “Introduction to Programming”
- For each test
  - Run the program, check output
  - But you didn’t think about correct output ahead of time

Testing in Vandy CS Courses

- CS 2201: Program Design and Data Structures
- Given compiled correct solution, compare your solutions I/O to the correct solution's
- For each test
  - Pipe input to correct solution, save output
  - For each buggy solution
    - Pipe input to buggy solution, diff output with result from correct solution
    - If outputs differ, a bug is exposed!

Testing in Vandy CS Courses

- CS 2201: Program Design and Data Structures
- Given unit tests
- For each test
  - Run test against correct solution
    - Throw out the test if it fails
  - For each buggy solution
    - Run test against buggy solution
    - If assertion fails, a bug is exposed!

Discussion: Vandy CS Testing

- Consider: What are the pros and cons of each?
- Recall
  - 1100/1101: 1 `main()` function == 1 test; check output
  - 2201: 1 input file == 1 test; output diff
  - 2201: 1 function with `assert()` == 1 test; assertion failure

Testing: Inputs and Outputs

- For 1100/1101, students write program inputs, but not expected outputs
- For 2201, students write program inputs and also expected outputs
- In real life, you rarely have an already-correct implementation of your program
- Testing with random inputs (**fuzz testing**) can help detect "bad things" bugs (segfaults, memory errors, crashes, etc.)
  - But does not provide full expected outputs

Testing Concepts

- Regression Testing
- Unit Testing
- XUnit
- Test-Driven Development
- Integration Testing
- Mocking
- Fuzz testing
- Penetration testing

Regression Testing (in one slide)

- Have you ever had one of those “I swear we've seen and fixed this bug before!” moments?
  - Perhaps you did, but someone else broke it again
  - This is a **regression** in the source code
- **Best practice**: when you fix a bug, add a test that specifically exposes that bug
  - This is called a **regression test**
  - It assesses whether future implementations still fix the bug

Regression Testing Story

```
// Dear maintainer:
//
// Once you are done trying to 'optimize' this routine,
// and have realized what a terrible mistake that was,
// please increment the following counter as a warning
// to the next guy:
//
// total_hours_wasted_here = 42
```

https://stackoverflow.com/questions/184618/what-is-the-best-comment-in-source-code-you-have-ever-encountered/482129#482129

Unit Testing and Frameworks

- In **unit testing**, “individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.”
- Modern frameworks are often based on SUnit (for Smalltalk), written by Kent Beck
  - Java JUnit, Python unittest, C++ googletest, etc.
- These frameworks are collectively referred to as **xUnit**

xUnit Features

- Test cases “look like other code”
  - They are special methods written to return a boolean or raise assertion failures
- A test case discoverer finds all such tests
- A test case runner chooses which tests to run

xUnit Definitions

- In xUnit, a test case is
  - A piece of code (usually a method) that
    - establishes some preconditions
    - performs an operation
    - and asserts postconditions
      - May give helpful error messages
- A **test fixture**
  - Specifies code to be run before/after each test case
  - Each test is run in a “fresh” environment

Python unit test Example

```
import unittest
class NiceThing:
    def __init__(self, num_spams):
        self.num_spams = num_spams
    def zap(self):
        return self.num_spams + 42


class NiceThingTestCase(
         unittest.TestCase):
    def setUp(self):
        self.nice_thing = NiceThing(0)
    def test_zap(self):
        self.assertEqual(45, self.nice_thing.zap())

if __name__ == '__main__':
    unittest.main()
```

```
$ python3 unit_test_demo.py
F
======================================================================
FAIL: test_zap (__main__.NiceThingTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unit_test_demo.py", line 11, in test_zap
    self.assertEqual(45, self.nice_thing.zap())
AssertionError: 45 != 42

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

- See Python unittest documentation for more details:
  - https://docs.python.org/3/library/unittest.html

Unit Testing Advantages

- Unit testing tests features in isolation
  - In the previous example, our test for `zap()` tested only the `zap()` method
  - Advantage: when a test fails, it is easier to locate the bug
- Unit testing tests are small
  - Advantage: smaller test are easier to understand
- Unit testing tests are fast
  - Advantage: fast tests can be run frequently

Test-Driven Development

- "**Test-driven development** is a software development process that relies on the **repetition** of a very **short** development **cycle**: *requirements* are turned into very *specific test cases*, then the software is improved so that *the tests pass*."
- Write a unit test for a new feature
  - When you run the test, it should fail
- Write the code that your unit test case tests
- Run all available tests
  - Fix anything that breaks; repeat until no tests fail
- Go back to step 1

Integration Testing

- Typically, any feature can be made to work in isolation
- What happens when we put our unit-tested features together into a larger program?
- Does our application work from start to finish?
  - “End-to-end” testing
- **Integration testing** combines and tests individual software modules as a group.

Integration Testing Examples

- Integration testing is application-specific
- CS Classes
  - Run main program with input file
- Web and GUI Applications
  - Use a testing framework (or harness) that lets you simulate user clicks and other input
- Systems Software
  - Use a testing framework that lets you simulate disk and network failures (cf. Chaos Monkey later)

Creative Integration Testing Examples

- For video games, you might write an AI to play
  - Bayonetta https://www.platinumgames.com/official-blog/article/6968
- Or have players use gaze-detecting goggles

https://www.tobiipro.com/fields-of-use/user-experience-interaction/game-usability/

“We see … modern eye tracking technology as a future standard in modern QA teams to improve the overall quality of game experiences.”

- Markus Kassulke, CEO, HandyGames

Psychology: Confirmation Bias

- **Confirmation bias** is the tendency to search for, interpret, favor, and recall information in a way that affirms one's prior beliefs or hypotheses. It includes a tendency to test ideas in a one-sided way, focusing on one possibility and ignoring alternatives.
- It is so well-established that experimental evidence is available in many flavors
- [R Nickerson. (1998). Confirmation Bias: A Ubiquitous Phenomenon in Many Guises. In Review of General Psychology, 2(2):175-220.]

Psychology: Confirmation Bias
(each subclaim has its own studies)

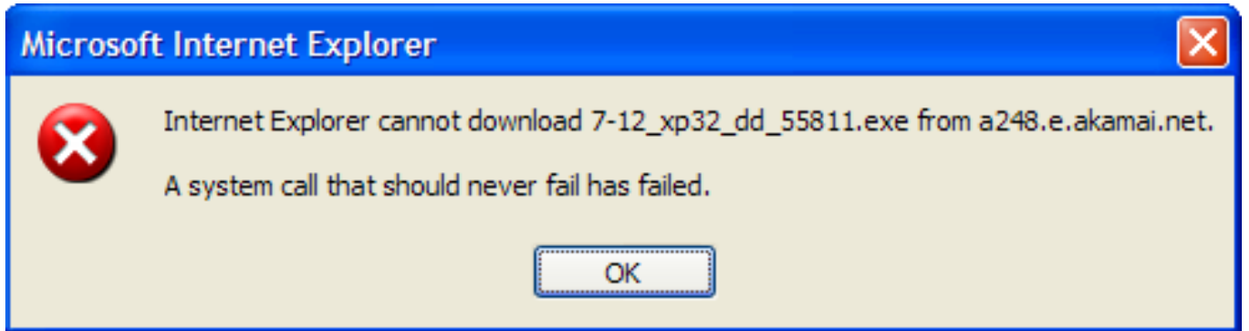
- Restriction of attention to a favored hypothesis
- Preferential treatment of evidence supporting existing beliefs
- Looking only or primarily for positive cases
- Overweighting positive confirmatory instances
- Seeing what one is looking for
- Favoring information acquired early

Psychology: Confirmation Bias

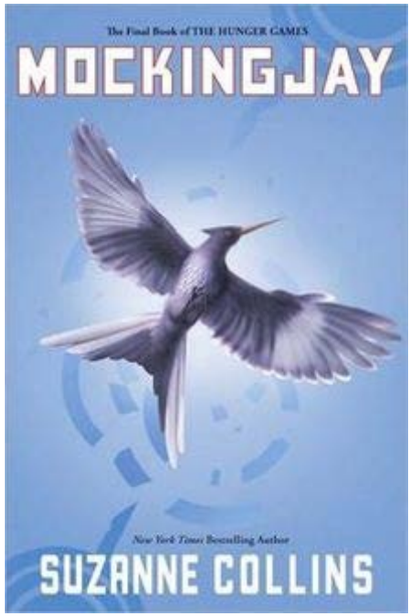
- Implications for SE:
- **Policy Rationalization** justifies policies to which an organization has already committed. “Once a policy has been adopted and implemented, all subsequent activity becomes an effort to justify it.”
- **Theory Persistence** involves holding to a favored idea long after the evidence against it has been sufficient to persuade others who lack vested interests.
- Idea or policy = any SE process decision.

Targeting Hard-To-Test Aspects

- What if we want to write unit or integration tests for some module/function/class, but it has expensive dependencies?
- Discuss: What are examples of things that are hard to test because they require extensive dependencies or entail too much cost?



Mocking



- "**Mock objects** are simulated objects that mimic the behavior of real objects in controlled ways."
- In testing, **mocking** uses a mock object to test the behavior of some other object.
  - Analogy: use a crash test dummy instead of real human to test automobiles

Scenario 1: Web API Dependency

- Suppose we're writing a single-page web app
- The API we'll use (e.g., Speech to Text) hasn't been implemented yet or costs money to use
- We want to be able to write our frontend (website) code without waiting on the server-side developers to implement the API and without spending money each time
- What should we do?

Mocking Dependencies

- Solution: make our own “fake” (“mock”) implementation of the API
- For each **method** the API exposes, write a **substitute** for it that just *returns some hard-coded data* (or any other approximation)
- This technique was used to design and test parts of the autograder website

Scenario 2: Error Handling

- Suppose we're writing some code where certain kinds of errors will occur sporadically once deployed, but “never” in development
  - Out of memory, disk full, network down, etc.
- We'd like to apply the same strategy
  - Write a fake version of the function …
- But that sounds difficult to do manually
  - Because many functions would be impacted
  - Example: many functions use the disk

Mocking Libraries: Two Approaches

- Before running the program (“static”)
  - Combine modularity/encapsulation with mocking
  - Move all disk access to a wrapper API, use mocking there at that one point (coin flip → fake error)
- While running the program (“dynamic”)
  - While the program is executing, have it rewrite itself and replace its existing code with fake or mocked versions
  - Let's explore this second option in detail

Dynamic Mocking Support

- Some languages provide **dynamic mocking libraries** that allow you to substitute objects and functions at runtime
  - For one test, we could use a mocking library to force another line of code inside our target function to throw an exception when reached
- This feature is available in modern dynamic languages (Python, javascript, etc.)
  - GoogleTest used to require a special base class for this sort of mocking, now it uses macros (for C++)

Dynamic Mocking Example

```
import unittest
from unittest import mock

def lowLevelOp():
  # might fail for users
  # example: no memory
  pass

def highLevelTask():
  try:
    lowLevelOp()
    return True
  except MemoryError:
    return False
```

```
class HLTTestCase(unittest.TestCase):
  def test_LLO_no_memory(self):
    def mocked_memory_error():
      raise MemoryError('test :-(')

    with mock.patch( # look here!
          '__main__.lowLevelOp',
          mocked_memory_error ):
      self.assertFalse(highLevelTask())

if __name__ == '__main__':
  unittest.main()
```

See https://docs.python.org/3/library/unittest.mock.html

See https://docs.python.org/3/library/unittest.mock.html#patch

Dynamic Mocking Disadvantages

- Test cases with dynamic mocking can be very fragile
  - What if someone moves or removes the call to `lowLevelOp()` that we `mock.patch`'d earlier?
- Dynamic mocking requires good integration tests
  - If we mock dependencies, we need to be extra careful that our ADTs play nicely together
- Dynamic mocking libraries have a learning curve
  - In Python, it can be hard to determine the correct value for '`path`' in `mock.patch` (etc.)
  - Error messages are often cryptic (modified program)

Fuzz Testing (Fuzzing)

- How can we generate many different inputs fast?
- Input massive amounts of random data ("fuzz"), to the test program in an attempt to make it crash/expose bad behavior

Fuzz Testing (Fuzzing)

- Barton Miller, University of Wisconsin, 1989
  - A night in 1988 with thunderstorm and heavy rain
  - Connected to his office Unix system via a dial up connection
  - The heavy rain introduced noise on the line
  - Crashed many UNIX utilities he had been using everyday
  - He realized that there was something deeper
  - Asked three groups in his grad-seminar course to implement this idea of fuzz testing:
    - Two groups failed to achieve any crash results!
    - The third group succeeded! Crashed 25-33% of the utility programs on the seven Unix variants that they tested

Fuzz Testing (Fuzzing)

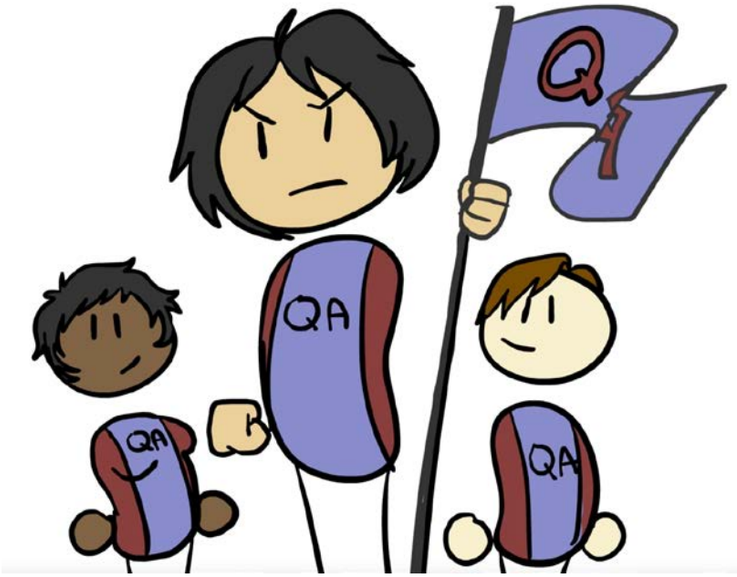
- Approach
  - Generate random inputs
  - Run lots of programs using random inputs
  - Identify crashes of these programs
  - Correlate random inputs with crashes
  - Errors found: Not checking returns, Array indices out of bounds, not checking null pointers, …
- American Fuzzy Lop (AFL) ---> HW2!!
  - Fuzzing by applying various modifications to the input file

Penetration Testing (Pen Testing)

- Security-oriented testing
  - Typically performed on a whole IT system, not just a single program
- Good intentioned
  - Performed by white hackers
  - With the goal of reporting found vulnerabilities
  - Can be part of a security audit
- National Cyber Security Center definition:
  "A method for gaining assurance in the security of an IT system by attem pting to breach some or all of that system's security, using the same tool s and techniques as an adversary might."

Quality Assurance and Development Processes

- How can we assure quality before, during and after writing code?
- What if we don't have enough resources?
  - Tune in next time!
- Further Watching:
  - “So You Want To Be In QA?”
  - https://www.youtube.com/watch?v=ntpZt8eAvy0



Questions?

- Next exciting episode:
  - Test Suite Quality *Metrics*
- HW1a due this Sunday
- You should email me if you are not on Piazza and/or autograder